Caracas, 19 de abril de 2023

Universidad Simón Bolívar

Diseño de Algoritmos 2

Profesor: Ricardo Monascal

Estudiante: Luis Alexander Parra Espinoza

Cobertura Mínima de Vértices

Índice

Introducción

Desde los inicios desde la computación, el ser humano se ha encargado de buscarle solución a muchos de los problemas existentes hoy en día, esto se ha logrado por medio de estudios de los comportamientos de algunos problemas, logrando así conseguir acotarlos en un tiempo comprensible.

Aún así existen problemas que crecen con una magnitud tal, que incluso sabiendo como conseguirle solución el procesamiento de estos problemas podrían tardar incluso años, estos son los llamados “problemas NP” con NP refiriéndose a No polinomiales, en palabras simples, la curva que acota su tiempo de ejecución no es Polinómica, es decir, es acotada por una función exponencial.

Viendo que buscar una solución exacta tomaría mucho tiempo, lo que se ha hecho para poder conseguir la información en un tiempo aceptable es conseguir una aproximación, y eso es donde entra este documento, nos encargaremos de probar diferentes técnicas para atacar el problema de **Cobertura Mínima de Vértices** o por sus siglas en ingles **MVC (Minimum Vertex Cover)**, sin más que decir, disfruten de la lectura.

Desarrollo

Primero que nada, hay que saber ¿Qué es un MCV? Supongamos tenemos un grafo no dirigido G= (V, E) donde V es la cantidad de vértices del grafo y E es la cantidad de lados o aristas del grafo, el MCV sería un Grafo G' tal que G'= (V', E), donde V' ⊆ V, es decir que el MCV sería la cantidad mínima de vértices necesarias para que sin importar cual arista perteneciente a E elijamos, ambos vértices de dicha arista pertenecen a V'.

Con esto en mente primero se comenzó por elegir un lenguaje que fuera bastante rápido para poder hacer pruebas constantemente, en este caso se opto por utilizar C++ como herramienta para las pruebas realizadas en este documento, esto corriendo Ubuntu en WSL (Windows Subsystem Linux) y utilizando Visual Studio Code como IDE para programar, por ultimo se creo un repositorio en Github para llevar un control de la información, así como un respaldo para la información.

A nivel de que algoritmos utilizamos, fueron algunos vistos en las clases de **Diseño de Algoritmos 2** en la Universidad Simón Bolívar, dictada este trimestre por el profesor Ricardo Monacal, entre estos encontramos: Búsqueda Local, Búsqueda Local Iterada, Búsqueda Taboo, Algoritmo Genetico y Memetico, entre otros. Algunos algoritmos se adaptan mejor que otras en ciertas situaciones, a su vez como otros son muy dependientes de otros algoritmos para dar una solución más eficiente. Si bien algunos métodos pueden ser mejores que otros en este problema en concreto, no significa que el método de por si sea malo, sino que o el modelado provisto no fue el más adecuado o simplemente ese tipo de estrategias no funciona muy bien para este caso.

El desarrollo se llevó con un ritmo un poco cuesta arriba ya que todo fue desarrollado por una persona, no implica que sea menos interesante solo que hubo más trabajo por realizar y mucho tiempo dedicado a analizar el como hacer funcionar algunos de los métodos que se especificaran más adelante. No obstante, me divertí planteando algunos de los métodos utilizados, además de llevarme unas sorpresas inesperadas. Algunas de las optimizaciones hechas fueron en base a información recopilada de otras fuentes de información que, como yo, decidieron atacar el problema de MVC para hallar un método mucho más rápido y con resultados iguales o mejores.

El Benchmark utilizado en este paper es el Benchmark de DIMACS, que son los mismos utilizados tanto para el MVC y MC (Maximun Cliqué), y las coberturas exactas fueron adquiridas en base a la información recopilada por otros documentos, ya que en la página no están los valores de las coberturas de los casos de prueba.

Método Exacto

Si bien como dijimos antes existen maneras de conseguir la respuesta que necesitamos de manera exacta, esta podría tardar una cantidad de tiempo abismal ya que estamos tratando con un problema NP, aun sabiendo eso, se diseñó un método para conseguir una solución exacta, aunque es muy, pero muy lento. El método utilizado es back tracking, partiendo del conjunto de todos los vértices, y se prueba ir quitando vértice por vértice de manera recursiva, hasta conseguir el mínimo posible, esta solución tarda mucho tiempo tanto así que no tengo los tiempos para mostrar en una tabla.

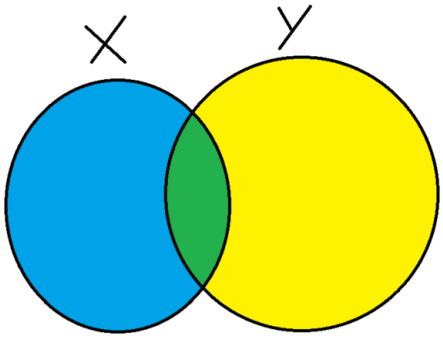
Aun así, tratando de plantear un algoritmo exacto por fuera del backtracking genere un algoritmo greedy que aproxima una solución decente en unos tiempos bastante rápidos. El principio básico usado para este método fue suponer que los vértices con mayor cantidad de aristas deben estar presentes en la cobertura mínima, a su vez se usaron algunas estrategias descritas en un documento sobre problemas NP por parte de DIMACS (enlace en la bibliografía). Con esa combinación se obtuvieron los siguientes resultados.

Algoritmo de Aproximación Voraz				
Nombre	**Vértices Totales**	**Cobertura**	**Resultado**	**Tiempo (Seg.)**
C125.9	**125**	**91**	**92**	**~0.00014**
C250.9	**250**	**206**	**212**	**~0.0002**
C500.9	**500**	**<=443**	**452**	**~0.0012**
C1000.9	**1000**	**<=932**	**948**	**~0.0045**
Brock200_2	**200**	**188**	**192**	**~0.001**
Brock200_4	**200**	**183**	**185**	**~0.0003**
Brock400_2	**400**	**371**	**380**	**~0.0012**
Brock400_4	**400**	**367**	**381**	**~0.0014**
Brock800_2	**800**	**776**	**786**	**~0.01**
Brock800_4	**800**	**774**	**785**	**~0.017**

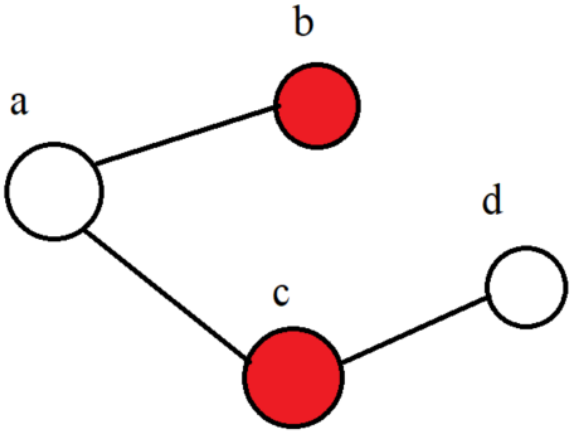
Como podemos observar en la tabla, la ejecución no tarda ni 1 segundo, lo cual lo hace sumamente rápido. Aunque hay casos donde se desvía bastante, por ejemplo, para el brock400_4, se podría decir que da un resultado bastante aceptable. Haciendo pruebas los tiempos de obtener un resultado de manera aleatoria son mayores que esto (aunque depende mucho de la implementación).

Búsqueda Local

La búsqueda local se fundamenta en definir una vecindad para cada elemento del conjunto solución, una vecindad es definir criterio en el cual pueda agrupar los elementos en sub conjuntos y poder llamarlos vecindad, que conste que las vecindades de cada elemento no son iguales, pero pueden tener elementos similares. Supongamos que tenemos Vx y Vy donde son la Vecindad de X y la Vecindad de Y correspondientemente, en la foto vemos como los conjuntos son diferentes, pero se interceptan para algunos elementos dado algún criterio.



Para nuestro trabajo estamos tomando como vecindad de un vértice, aquellos vértices que están conectados a dicho vértice, es decir, si tenemos los vértices a, b, c, y d donde a está conectado a b y c, la vecindad de a sería igual a los elementos b y c, Va = {b,c}.



Ya sabiendo nuestra vecindad, lo que haremos es que por cada vértice del conjunto solución, lo sustituiremos por algún elemento de la vecindad, siempre y cuando cambiar el elemento no vaya en contra del MVC, es decir que, si al quitar X y colocar Y resulta que no estoy cubriendo una arista, entonces ese cambio no es válido. Para que haya variación a la hora de hacer los movimientos, se van eligiendo los posibles candidatos de manera aleatoria hasta que alguno cumpla los requisitos necesarios para hacer el intercambio. Existe otro método que implica

verificar a todos los vecinos de un elemento dado para ver cual es el mejor candidato a ser intercambiado, pero, este proceso en muchas ocasiones puede tomar mucho tiempo por lo que se descartó.

Para este método, es necesario otro algoritmo que calcule una solución inicial a la que le será aplicada el método de optimización de búsqueda local. En nuestro caso empleamos el algoritmo voraz generado de manera no intencional como base para la optimización, aquí una tabla comparativa con la optimización y los tiempos

Búsqueda Local					
Nombre	**Vértices Totales**	**Cobertura**	**Resultado Voraz**	**Resultado Bus. Local**	**Tiempo (Seg.)**
C125.9	**125**	**91**	**92**	**92**	**~0.0007**
C250.9	**250**	**206**	**212**	**210-212**	**~0.0045**
C500.9	**500**	**<=443**	**452**	**450-452**	**~0.05**
C1000.9	**1000**	**<=932**	**948**	**942-947**	**~0.5**
Brock200_2	**200**	**188**	**192**	**191-192**	**~0.03**
Brock200_4	**200**	**183**	**185**	**185**	**~0.006**
Brock400_2	**400**	**371**	**380**	**377-380**	**~0.12**
Brock400_4	**400**	**367**	**381**	**377-381**	**~0.09**
Brock800_2	**800**	**776**	**786**	**783-785**	**~2.4**
Brock800_4	**800**	**774**	**785**	**784-785**	**~3.2**

Como podemos observar si se pueden optimizar los resultados obtenidos por la búsqueda voraz, si bien no son cambios drásticos, esto se debe a que el algoritmo voraz realizó la búsqueda de manera de conseguir lo mínimo posible según su criterio por ende esta fijado a un patrón especifico, si por ejemplo usáramos un algoritmo aleatorio y luego le aplicáramos búsqueda local al resultado obtenido seguramente podríamos observar mucho más fácil el impacto que realmente tiene la Búsqueda Local.

Búsqueda Local Iterada

La búsqueda local iterada, como bien su nombre dice consiste aplicar la Búsqueda Local anteriormente explicada múltiples veces, con la diferencia en que antes de cada Búsqueda Local, le realizamos una "Perturbación" a la solución. Esta perturbación consiste en modificar el conjunto solución con la finalidad de explorar otras regiones donde quizás sea posible que se encuentre la mejor solución del problema, la perturbación no tiene por qué generar una solución valida, solo la usaremos como un nuevo punto de partida para que la Búsqueda Local pueda intentar encontrar mejores resultados con un diferente punto de partida. Además de esto se posee una memoria con todas las perturbaciones realizas, esta se tiene para evitar volver a realizar Búsquedas Locales desde la misma perturbación inicial.

En mi caso, me dio por explorar una perturbación algo peculiar:

1. Una perturbación de tamaño variable, es decir que no siempre perturbo la misma cantidad de elementos.
2. Se perturban los vértices del conjunto solución con similitud de grados, esto lo hago ordenando la solución por grados, luego buscamos la mitad de la lista y vamos escogiendo desde la mitad para ambos lados de arreglo por igual para realizar la perturbación.

Este algoritmo a su vez que su predecesor se basa en una solución ya existente por lo que se volvió a usar el método voraz como solución inicial. Veamos los resultados.

Búsqueda Local Iterada						
Nombre	**Vértices Totales**	**Cobertura**	**Resultado Voraz**	**Resultado Bus. Itera.**	**Iteraciones**	**Tiempo (Seg.)**
C125.9	**125**	**91**	**92**	**~92**	**30**	**~0.007**
C250.9	**250**	**206**	**212**	**~211**	**50**	**~0.1**
C500.9	**500**	**<=443**	**452**	**~451**	**100**	**~1.3**
C1000.9	**1000**	**<=932**	**948**	**~945**	**100**	**~16**
Brock200_2	**200**	**188**	**192**	**~191**	**50**	**~0.3**
Brock200_4	**200**	**183**	**185**	**~185**	**50**	**~0.15**
Brock400_2	**400**	**371**	**380**	**~379**	**100**	**~2.3**
Brock400_4	**400**	**367**	**381**	**~381**	**100**	**~2**
Brock800_2	**800**	**776**	**786**	**~783**	**100**	**~54.4**
Brock800_4	**800**	**774**	**785**	**~785**	**100**	**~60**

Como podemos observar, los resultados no cambian mucho con respecto a los de la búsqueda local normal, esto se debe directamente a la solución inicial permite muy pocas optimizaciones, aun así, podemos ver que consigue resultados aceptables para una gran cantidad de iteraciones. Las filas resaltadas en un verde corresponden a las filas que tienen los grafos más pesados de las pruebas, estos grafos tienen muchas aristas, por ende, la diferencia notable en los tiempos de corrida, impacta más la cantidad de lados que la cantidad de vértices.

Búsqueda Tabú

La búsqueda tabú tiene un comportamiento muy similar a la búsqueda local iterada, solo que nos encontramos con un par de diferencias clave:

- La memoria acá es temporal, pero impide hacer movimientos que ya se hicieron de antemano, esto se hace para evitar acciones cíclicas.
- La Búsqueda local permite hacer movimientos que lleven a soluciones no validas

A la hora de desarrollar el algoritmo de búsqueda tabú se planteó utilizar una memoria temporal de tamaño N, donde N es el tamaño de la solución, para cada cambio hecho. Además, en este caso en vez de realizar cambios a todos miembros del conjunto solución se realizará a solo 1, elegido de manera al azar, esto se hace para evitar que se dañe completamente la solución impidiendo así llegar a una respuesta valida. Sumado a lo anterior, dado que el poco a poco la solución original se va corrompiendo, cada cierto numero de iteraciones, se restaura la solución tabú con la mejor solución encontrada hasta el momento y se sigue buscando. Para este caso, tomamos como solución inicial, una generada de manera aleatoria, observemos los resultados y analicemos.

Búsqueda Tabú					
Nombre	**Vértices Totales**	**Cobertura**	**Iteraciones**	**Resultado Bus. Tabú**	**Tiempo (Seg.)**
C125.9	**125**	**91**	**2000**	**~106**	**~0.1**
C250.9	**250**	**206**	**2000**	**~232**	**~0.3**
C500.9	**500**	**<=443**	**2000**	**~478**	**~1**
C1000.9	**1000**	**<=932**	**2000**	**~982**	**~5**
Brock200_2	**200**	**188**	**2000**	**~196**	**~0.95**
Brock200_4	**200**	**183**	**2000**	**~194**	**~0.6**
Brock400_2	**400**	**371**	**2000**	**~390**	**~1.9**
Brock400_4	**400**	**367**	**2000**	**~391**	**~2**
Brock800_2	**800**	**776**	**2000**	**~793**	**~18**
Brock800_4	**800**	**774**	**2000**	**~793**	**~18**

Podemos notar al ver la tabla que los resultados se alejan bastante de lo que debería ser la cobertura, incluso pareciera que no hiciera nada. El problema aquí cae en el diseño de la memoria. Al ser una cantidad tan grande de elementos y cada uno teniendo una variedad inmensa de posibles movimientos, la memoria tabú planteada inicialmente es casi inútil, para mejorar esto

se debería plantear que cada vértice posea una memoria temporal que se vaya eliminando progresivamente si es que se usa el mismo vértice de manera constante. Ya que al tener una memoria global de movimientos lo que ocurre es que es muy probable que la memoria se borre antes de que vea un uso, y si hacemos la memoria demasiado grande restringiríamos demasiado el movimiento de la búsqueda. Con ese cambio, los tiempo se mantendrían bastante competentes y los resultados, seguramente mejorarían drásticamente.

Algoritmo Genético

El algoritmo Genético esta inspirado en el comportamiento genético del ser humano, simulando tanto un genotipo como un fenotipo, donde el genotipo sería como un tipo de codificación de la solución, mientras que el fenotipo sería la solución concretamente dada. Definido ese comportamiento tomamos una población, y nuestro objetivo es simular los cambios poblacionales dentro de un grupo por medio de herencia y mutación. La herencia viene dada por los padres (un grupo de la población elegida para generar decendencia en una generación), otorgándole a sus hijos partes de sus genotipos, y la mutación puede ocurrir y generar un cambio adicional al genotipo del hijo. Por ultimo se seleccionan candidatos poblacionales que sobrevivirán para la próxima generación.

Para esta recreación de este algoritmo se tomaron:

- Una población de 90
- Entre 20-40 padres por generación
- Un genotipo de la siguiente forma, supongamos que nuestra solución de un grafo de 10 vértices es {1,2,7,9,10}, entonces su GENOTIPO sería un arreglo de booleanos de tamaño 10, donde la casilla será verdadera si su índice esta dentro del conjunto solución, de lo contrario será falso.

- Una probabilidad de mutación del 40% (Donde la mutación solo cambia 1 bit aleatorio del Genotipo)
- Un 70% de Mantener a los mejores de la población y 30% a los peores.
- Los Hijos heredan la mitad de los bits de cada padre.

A continuación, veamos los resultados para estas especificaciones:

Algoritmo Genético						
Nombre	**Vértices Totales**	**Cobertura**	**Iteraciones**	**Mutación**	**Resultado**	**Tiempo (Seg.)**
C125.9	**125**	**91**	**500**	**40%**	**~101**	**~2.2**
C250.9	**250**	**206**	**500**	**40%**	**~229**	**~10**
C500.9	**500**	**<=443**	**500**	**40%**	**~477**	**~60**
C1000.9	**1000**	**<=932**	**500**	**40%**	**~976**	**~805.05**
Brock200_2	**200**	**188**	**500**	**40%**	**~194**	**~28**
Brock200_4	**200**	**183**	**500**	**40%**	**~189**	**~19**
Brock400_2	**400**	**371**	**500**	**40%**	**~387**	**~111.2**
Brock400_4	**400**	**367**	**500**	**40%**	**~387**	**~106**
Brock800_2	**800**	**776**	**500**	**40%**	**~791**	**~2300**
Brock800_4	**800**	**774**	**500**	**40%**	**~791**	**~2270**

Al ver los resultados obtenidos es un poco desconcertante que los resultados se alejen bastante de la cobertura mínima, sinceramente no sé exactamente porque ocurre esto. Se hicieron pruebas con menores porcentajes de mutación y no variaba mucho el resultado, a si como se probó a cambiar una cantidad mayor de bits del genotipo a la hora de mutar y resultaba en resultados peores. Es probable que haya un problema de diseño más que impide que este tipo de comportamiento especifico triunfe de manera general. Es el algoritmo que más ajustes ha recibido con diferencia, más no logro hacerlo brillar más allá de lo que se tiene actualmente.

Algoritmo Memetico

El algoritmo memetico es muy similar al genético, en el sentido generacional, tener una población que se va actualizando y mutando, con un conjunto de padres y manteniendo ciertos comportamientos. Más no sigue todos los principios. En nuestro caso sigue los siguientes:

- Posee 4 padres elegidos de manera aleatoria (pueden ser los mismos padres)
- Se generan 4 hijos
  - Hijo1, es la unión de los valores de los genotipos de todos los padres.
  - Hijo2, es la negación del hijo 1.
  - Hijo3, es la intersección de los valores de los genotipos de todos los padres.
  - Hijo4, es la negación del hijo 3.
- No existe mutación directa, solo la generada por la herencia.
- La nueva población son directamente los hijos, no hay un proceso de selección riguroso
- Se le aplica una optimización de Búsqueda Local en cada generación a cada hijo.
- Conjunto inicial generado de manera aleatoria

Algoritmo Memetico					
Nombre	**Vértices Totales**	**Cobertura**	**Iteraciones**	**Resultado**	**Tiempo (Seg.)**
C125.9	**125**	**91**	**500**	**~94**	**~6.5**
C250.9	**250**	**206**	**500**	**~212**	**~47**
C500.9	**500**	**<=443**	**500**	**~456**	**~470**
C1000.9	**1000**	**<=932**	**500**	**~943**	**~700**
Brock200_2	**200**	**188**	**500**	**~192**	**~298**
Brock200_4	**200**	**183**	**500**	**~187**	**~146**
Brock400_2	**400**	**371**	**500**	**~380**	**~962**
Brock400_4	**400**	**367**	**500**	**~379**	**~1000**
Brock800_2	**800**	**776**	**500**	**~784**	**~1417**
Brock800_4	**800**	**774**	**500**	**~783**	**~1546**

Este algoritmo tiene un tanto peculiar, más que nada por como funcionan los hijos 1 y 4 funcionan como soluciones de control para que no se quede estancada la búsqueda, mientras que los otros dos hijos ofrecen variedad. Si bien por si solo ese comportamiento no sería efectivo, al mezclarlo con una optimización de búsqueda local permite tener constantemente una indicación de hacia donde esta una posible solución. El único problema es que tiene unos tiempos bastante elevados en contraparte con muchos de los algoritmos vistos anteriormente.

Colonia de Hormigas

La Colonia de Hormigas es el algoritmo que mas problemas me ha causado, más que nada porque siento que no se adapta bien al problema a resolver o no se me ocurre concretamente como hacer el modelado. Este se basa en simular el recorrido de varias hormigas siguiendo un comportamiento muy simple, dejando un rastro de feromonas tras de si con la finalidad de poco a poco conseguir una solución final para cada hormiga.

Para este problema se usaron las siguientes especificaciones:

- Un índice de disipación de feromonas del 30% por movimiento
- Una cantidad de 40 hormigas
- Se mueven en dirección del vértice con mayor grado a su alrededor o la zona con mayores feromonas (No pueden repetir vértice, a menos que se no les quede otra opción).

Para estas especificaciones se obtuvieron los siguientes resultados:

Colonia Hormigas					
Nombre	**Vértices Totales**	**Cobertura**	**Hormigas**	**Resultado**	**Tiempo (Seg.)**
C125.9	**125**	**91**	**40**	**~116**	**~0.2**
C250.9	**250**	**206**	**40**	**~244**	**~1**
C500.9	**500**	**<=443**	**40**	**~490**	**~7.4**
C1000.9	**1000**	**<=932**	**40**	**~983**	**~75**
Brock200_2	**200**	**188**	**40**	**~197**	**~2.5**
Brock200_4	**200**	**183**	**40**	**~195**	**~1.5**
Brock400_2	**400**	**371**	**40**	**~396**	**~12.3**
Brock400_4	**400**	**367**	**40**	**~393**	**~11.28**
Brock800_2	**800**	**776**	**40**	**~797**	**~236**
Brock800_4	**800**	**774**	**40**	**~795**	**~220**

El problema con este algoritmo son 2 cosas:

1. Como plantear el recorrido de la hormiga
2. Que la hormiga se mueve de vértice en vértice, mientras que en el MVC no necesariamente todos los vértices forman una cadena

El punto 1 es más pensar un buen modelado, pero el 2 es algo más complejo, se podría usar un registro de camino intermitente (un turno si, un turno no) para así moverse de una manera que podría ser una solución del MVC, aún así, este es uno de los algoritmos que más me ha costado diseñar y pensar para resolver este problema.

La Pesca

La pesca es una metaheurística inventada por mí, con la finalidad de sirva como planteamiento general para que se pueda conseguir una solución a un problema dado. La pesca se define de la siguiente manera:

- Se posee un Estanque (Matriz NxM) en el cual se colocarán los posibles elementos del conjunto solución de manera aleatoria, llamémoslos cebos.
- Cada cebo posee un peso, este peso determinará que tan hundido estará dentro del estanque.
- Dentro del estanque existirán una cantidad K de peces, donde existirán varios tipos de peces (Colocare 3 casos, más esto puede variar a gusto del usuario)
    - Pez chiquito, prefiere los cebos que están menos profundos.
    - Pez mediano, prefiere los cebos que están medianamente profundos
    - Pez grande, prefiere los cebos que están más profundos.
- Los peces se moverán de manera aleatoria a lo largo del estanque, si ven un cebo optarán por picar o no, si pican el cebo es eliminado del estanque.
- La condición de parada será después de C ciclos.
- El conjunto de solución inicial serán los cebos picados por los peces, a este proceso se le aplicara un algoritmo de optimización (En el caso particular de este ejercicio, Búsqueda Local)
- Se realizar este algoritmo varías veces para obtener varios resultados y así conseguir una posible solución.

Mi pensamiento con este algoritmo es que puedes agregarle comportamientos diferentes por medio de los peces, teniendo una mayor cantidad de peces de un tipo especifico rondando por el estanque puede causar variaciones en el conjunto solución inicial que quizás permitan llegar a el óptimo del problema.

Para este caso en concreto se utilizaron las siguientes especificaciones:

- 200 ciclos
- El peso del cebo es el peso del vértice
- Los segmentos de altura están calculados en baso al cebo mas liviano y al cebo mas pesado.
- Un estanque de tamaño NxN donde N es el nro. de vértices
- 3 tipos de peces:
    - Pez chico, 50% picar cebo liviano, 20% picar cebo mediano, 10% picar cebo grande, 20% no hacer nada.
    - Pez mediano, 20% picar cebo liviano, 50% picar cebo mediano, 10% picar cebo grande, 20% no hacer nada.
    - Pez grande, 10% picar cebo liviano, 20% picar cebo mediano, 50% picar cebo grande, 20% no hacer nada.

- Una optimización usando Búsqueda Local.

Algoritmo Inventado (La Pesca)						
Nombre	**Vértices Totales**	**Cobertura**	**Iteraciones**	**Peces**	**Resultado**	**Tiempo (Seg.)**
C125.9	**125**	**91**	**200**	**2 * 125**	**~93**	**~0.9**
C250.9	**250**	**206**	**200**	**2 * 250**	**~214**	**~2.6**
C500.9	**500**	**<=443**	**200**	**4 * 500**	**~457**	**~17.6**
C1000.9	**1000**	**<=932**	**200**	**8 * 1000**	**~950**	**~153.05**
Brock200_2	**200**	**188**	**200**	**2 * 200**	**~192**	**~5**
Brock200_4	**200**	**183**	**200**	**2 * 200**	**~187**	**~4**
Brock400_2	**400**	**371**	**200**	**4 * 400**	**~380**	**~25**
Brock400_4	**400**	**367**	**200**	**4 * 400**	**~378**	**~26**
Brock800_2	**800**	**776**	**200**	**8 * 800**	**~784**	**~488**
Brock800_4	**800**	**774**	**200**	**8 * 800**	**~783**	**~495**

Con esos parámetros, estos fueron los datos recolectados:

Como podemos ver, tiene unos buenos valores con unos tiempos aceptables en comparación a los demás. Cabe destacar que las posibles soluciones oscilan dentro de lo que podría ser la cobertura mínima posible, más no termina de conseguirla ya que debe ser una selección especifica de vértices.

Comparación entre los Algoritmos

Comparación Resultados								
Nombre	**Greedy**	**BL**	**BLI**	**BT**	**Al. Gen.**	**Al. Meme**	**Col. Horm.**	**La Pesca**
C125.9	92	92	~92	~106	~101	~94	~116	~93
C250.9	212	210-212	~211	~232	~229	~212	~244	~214
C500.9	452	450-452	~451	~478	~477	~456	~490	~457
C1000.9	948	942-947	~945	~982	~976	~943	~983	~950
Brock200_2	192	191-192	~191	~196	~194	~192	~197	~192
Brock200_4	185	185	~185	~194	~189	~187	~195	~187
Brock400_2	380	377-380	~379	~390	~387	~380	~396	~380
Brock400_4	381	377-381	~381	~391	~387	~379	~393	~378
Brock800_2	786	783-785	~783	~793	~791	~784	~797	~784
Brock800_4	785	784-785	~785	~793	~791	~783	~795	~783

Comparación Tiempos (Seg.)								
Nombre	**Greedy**	**BL**	**BLI**	**BT**	**Al. Gen.**	**Al. Meme**	**Col. Horm.**	**La Pesca**
C125.9	~0.00014	~0.0007	~0.007	~0.1	~2.2	~6.5	~0.2	~0.9
C250.9	~0.0002	~0.0045	~0.1	~0.3	~10	~47	~1	~2.6
C500.9	~0.0012	~0.05	~1.3	~1	~60	~470	~7.4	~17.6
C1000.9	~0.0045	~0.5	~16	~5	~805.05	~700	~75	~153.05
Brock200_2	~0.001	~0.03	~0.3	~0.95	~28	~298	~2.5	~5
Brock200_4	~0.0003	~0.006	~0.15	~0.6	~19	~146	~1.5	~4
Brock400_2	~0.0012	~0.12	~2.3	~1.9	~111.2	~962	~12.3	~25
Brock400_4	~0.0014	~0.09	~2	~2	~106	~1000	~11.28	~26
Brock800_2	~0.01	~2.4	~54.4	~18	~2300	~1417	~236	~488
Brock800_4	~0.017	~3.2	~60	~18	~2270	~1546	~220	~495

Conclusión

A lo largo de este paper hemos visto varias técnicas para atacar el problema MVC, cada una tiene comportamientos diferentes y en ciertas circunstancias puede ser mejor o peor. Algo que acotar es que entre más aristas tenga es más probable que la cobertura mínima requiera más vértices. Muchos de los algoritmos utilizados se han visto con tiempos mayores más por la cantidad de aristas que por la cantidad de vértices del mismo, esto se puede ver en contraste con los brock800 y el C1000.9, el brock8000 tiene hasta 100000 aristas mientras que el C1000 apenas llega a la mitad de eso.

También podemos observar que muchos de los algoritmos que utilizaron Búsqueda Local tuvieron un rendimiento muy bueno o decente, esto debido a la misma naturaleza de la Búsqueda local de optimizar soluciones con la vecindad que se dijo casi al inicio del documento, es posible plantear otro tipo de vecindad, aunque esta en mi opinión es la de más facilidad y entendimiento para modelar el problema.

Para la búsqueda tabú como se menciono en su apartado, puede tener un rendimiento mucho mejor si le aplicamos es una memoria individual a cada vértice, ya que de esta manera es más rápido acceder a los movimientos no validos y podemos tener una constancia que si se utilizara la memoria como es debido, por falta de tiempo no se pudo hacer esa corrección, más es algo que seguramente tendrá un mejor impacto en el mismo.

El algoritmo genético realmente fue una sorpresa, pensé que tendría un mejor rendimiento del encontrado en esta implementación. Si bien aquí utilizamos una gran probabilidad para que se realice una mutación, la mutación solo cambia 1 bit del Genotipo, por lo que va progresivamente. Tal vez el problema que tenga actualmente en la selección de la nueva población, quizás un ajuste de valores sería mejor, el actual es 70% de agarrar una solución validad y 30% de que no, se podría verificar eso.

El algoritmo memetico es un meme en sí mismo, no pensé que fuera a tener un performance tan bueno, dado a que el genético no tuvo un buen rendimiento, aun así, creo que parte de esto se debe a que la búsqueda local esta ayudando bastante a conseguir soluciones aceptables. Igual es no quita que la solución proporcionada de manera inicial deber con la mínima decente para que esto ocurra.

La colonia de hormigas fue una decepción, creo que el método no se adapta bien para lo que queremos resolver, las hormigas como siguen de nodo a nodo no necesariamente van a escoger la mejor solución que se pueda, tampoco ayuda un poco lo complejo modelar el problema a este método, igual creo que si se podría hacer funcionar.

La pesca fue una sorpresa en si misma, me base un poco en buscar soluciones como una hormiga más viéndolo desde otra perspectiva y funcionó, si bien se le aplica una optimización a cada resultado, las posibles soluciones dadas son bastantes buenas, pero no terminan de ser una solución viable por lo que la optimización le termina de dar ese empujoncito necesario para brillar.

Por otra parte, el resto de los valores son completamente personalizables lo cual puede garantizar conseguir una variedad de resultados intentando diferentes combinaciones de parámetros y si se consigue modelar el problema de la manera correcta.

Ya unas ultimas palabras para agregar, si bien los problemas NP seguirán siendo grandes problemas hasta que se demuestre la incertidumbre de si NP=P o Np != P, podemos decir que son problemas que podemos afrontar buscando aproximaciones que si bien no son los resultados exactos, son lo suficientemente buenas para poder ser información confiable y segura para su uso. Espero hayan disfrutado leer esto tanto como yo lo disfrute programando y analizando todo esto. Gracias por leer.

Bibliografía

- http://archive.dimacs.rutgers.edu/Publications/Modules/Module06-1/dimacs06-1.pdf
- https://arxiv.org/ftp/arxiv/papers/1402/1402.0584.pdf
- https://www.researchgate.net/publication/242539177_Optimization_of_Unweighted_Minimum_Vertex_Cover
- https://drops.dagstuhl.de/opus/volltexte/2022/16546/pdf/LIPIcs-SEA-2022-12.pdf
- https://lcs.ios.ac.cn/~caisw/VC.html